LA-UR-21-30257
Approved for public release; distribution is unlimited.

Title: mystic: software for autonomous discovery and design under uncertainty

Author(s): McKerns, Michael

Intended for: CAMERA Workshop on Autonomous Discovery in Science and Engineering, 2021-04-20/2021-04-22 (Berkeley, California, United States)

Issued: 2021-11-04 (rev.1)

mystic: software for autonomous discovery, design, and control under uncertainty

mystic

a framework for highly-constrained
non-convex optimization
and uncertainty quantification

Mike McKerns

mystic: model validation and optimal design

- original funding: neutron instrument tuning & experiment design
- 20+ years of development with over 65$M of funding
  - DANSE (NSF): non-convex optimization and experiment design
  - PSAAP (NNSA): parallel/distributed computing and UQ/V&V
  - ExMatEx (ASC): scalability, reliability, and persistence
  - additional funding from DARPA, AFOSR, DTRA, LANL, BNL, ...
  - R&D/prod: JPMorgan, Barclays, Morgan-Stanley, LMCO, Roche, UTRC, ...

learn model robustness/accuracy under uncertainty

IS&T Theme #3: Machine Learning and Artificial Intelligence

"Development of reliable and trustworthy algorithms, methods and models to enable machine learning and artificial intelligence technologies for science and security."

Priorities: Specific areas of interest include:

"Integration into and optimization of experimental, computational, and observational workflows, including methods that combine heterogeneous data, or exploit small datasets."

This work will tackle the following themes:

- Incorporating physics domain knowledge into ML through physics-informed kernels and regularizers.
- Use UQ-driven active learning to produce optimally robust surrogates.
- Demonstrating reproducibility and well-posedness in ML surrogates.

Foundational research themes from BRN report

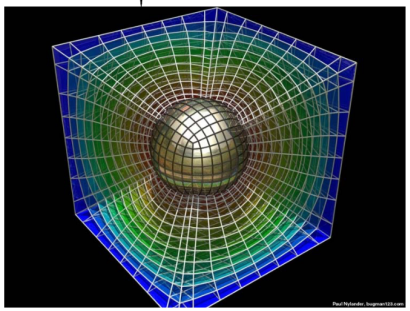
solve PDEs w/ digital transformation (automation)

$$\Delta u = f$$

A. L. Cauchy
(1789-1857)

S. D. Poisson
(1781-1840)

$$\Delta u = f$$



By the Herglotz integral formula, $f_n(z) = \int_0^{2\pi} \frac{e^{i\theta}+z'}{e^{i\theta}-z'} \operatorname{Re} f_n(ae^{i\theta}+b)\frac{d\theta}{2\pi}$,

So

$$|f_n(z)-f_n(w)| = \left|\int \left(\frac{e^{i\theta}+z'}{e^{i\theta}-z'} - \frac{e^{i\theta}+w'}{e^{i\theta}-w'}\right) \operatorname{Re} f_n(ae^{i\theta}+b)\frac{d\theta}{2\pi}\right|$$

$$\leq \int \frac{|e^{i2\theta} - e^{i\theta}w' + e^{i\theta}z' - z'w' - e^{i2\theta} + e^{i\theta}w' + e^{i\theta}z' + z'w'|}{|e^{i\theta}-z'|\,|e^{i\theta}-w'|} |\operatorname{Re} f_n| \frac{d\theta}{2\pi}$$

Since K is compact, $\exists \eta > 0$ such that $|ae^{i\theta}+b - (az'+b)| = a|e^{i\theta}-z'| \geq \eta$ $\forall z \in K$. Since $\{\operatorname{Re} f_n\}$ converges uniformly on ∂D, it is uniformly bounded there:

$$|f_n(z) - f_n(w)| \leq \frac{2|z'-w'|M_a}{\eta^2} = \frac{2M_a}{\eta^2}|z-w|.$$

can we automate model design and validation?

Where are we at in finding statistical estimators?

Find the
best estimator
or model

θ

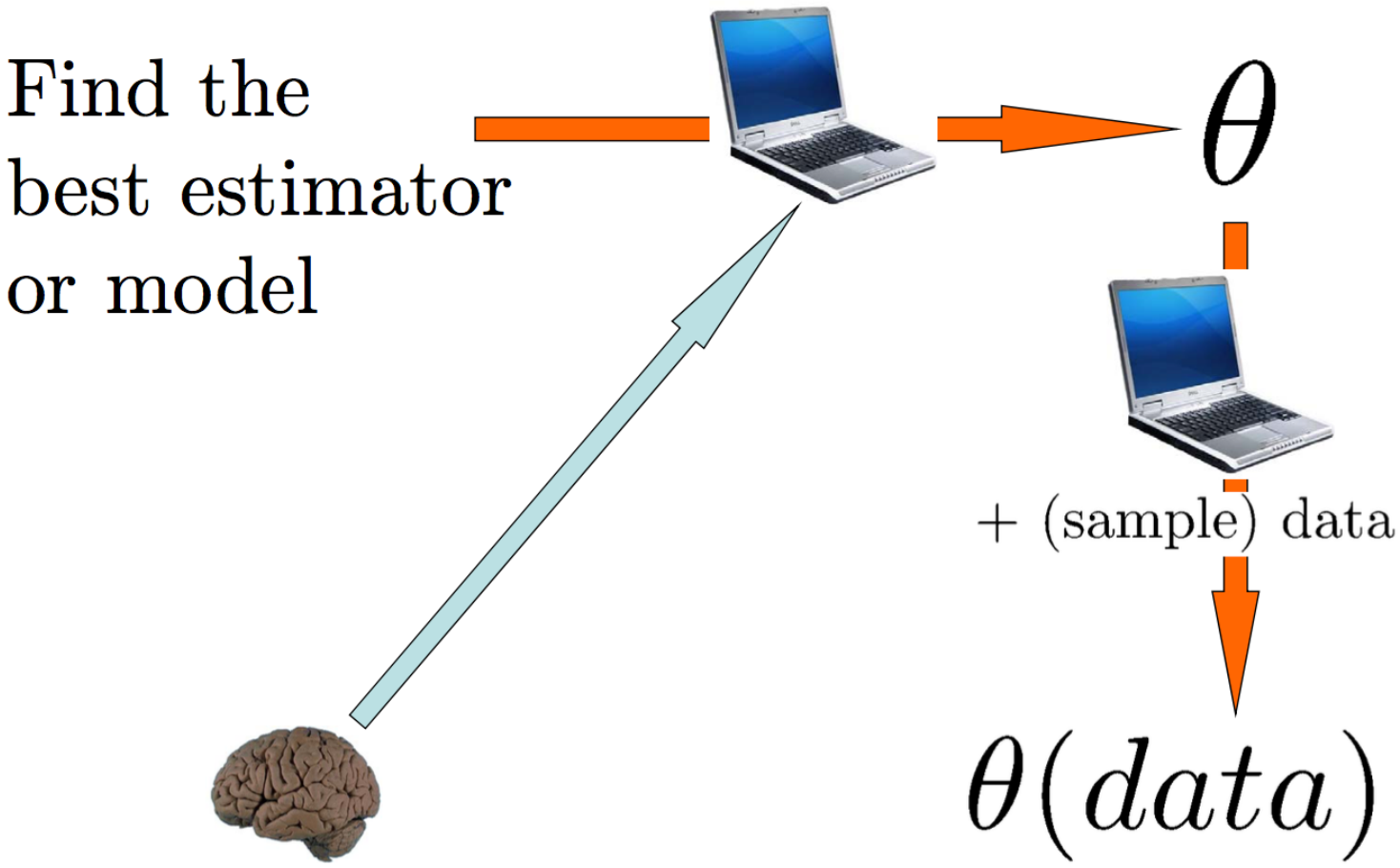
+ (sample) data

$\theta(data)$

designing an estimator is currently very laborious manual process

human intellect into design of the computation

Can we turn model design into a computation?



model certification/validation under uncertainty

The UQ challenge in the certification context
(Performance of a weapon system)

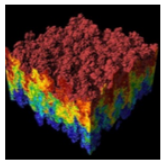
You want to certify that

$$\mathbb{P}[\text{failure}] \leq \text{treshold}$$

Problem

- You cannot test it.
- You don't know all possible causes of a failure
- You don't know $\mathbb{P}$

BUT

- You can simulate 
- You have 20 samples from the old system

state the problem in terms of what we know

You want to certify that

$$\mathbb{P}[G(X) \geq a] \leq \epsilon$$

Problem

- You don't know G.

and

- You don't know $\mathbb{P}$

You only know

$$(G, \mathbb{P}) \in \mathcal{A}$$

$$\mathcal{A} \subset \left\{ (f, \mu) \,\middle|\, \begin{array}{c} f : \mathcal{X} \to \mathbb{R}, \\ \mu \in \mathcal{P}(\mathcal{X}) \end{array} \right\}$$

compute bounds determined by what we know

Compute **Worst and best case**

optimal bounds $\mathbb{P}[G(X) \geq a]$
given available information.

$$\mathcal{U}(\mathcal{A}) := \sup_{(f,\mu)\in\mathcal{A}} \mu[f(X) \geq a]$$

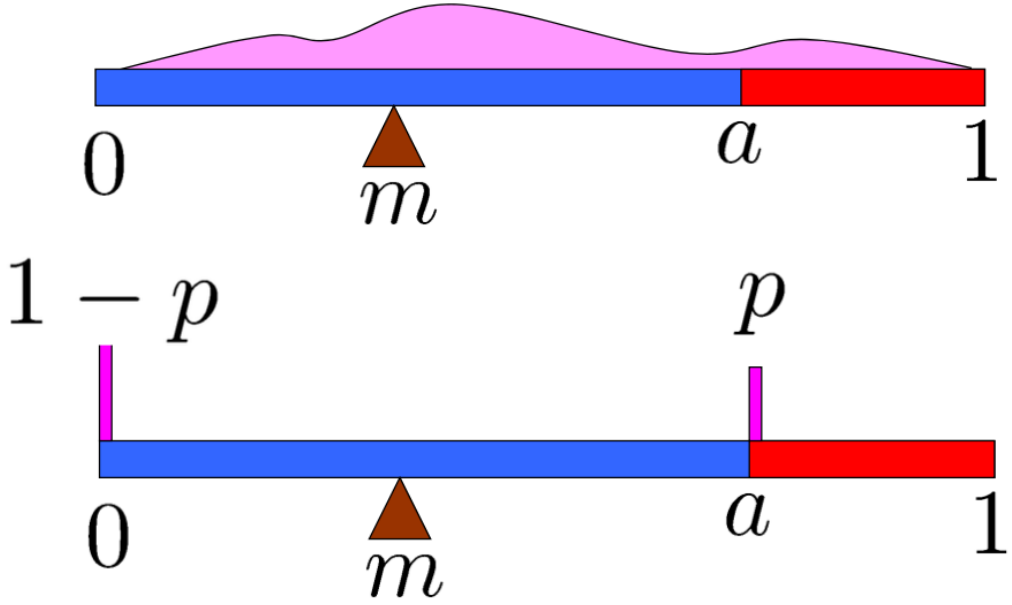
$$\mathcal{L}(\mathcal{A}) := \inf_{(f,\mu)\in\mathcal{A}} \mu[f(X) \geq a]$$

$$\mathcal{L}(\mathcal{A}) \leq \mathbb{P}[G(X) \geq a] \leq \mathcal{U}(\mathcal{A})$$

$\mathcal{U}(\mathcal{A}) \leq \epsilon$: Safe even in worst case.
$\epsilon < \mathcal{L}(\mathcal{A})$: Unsafe even in best case.
$\mathcal{L}(\mathcal{A}) \leq \epsilon < \mathcal{U}(\mathcal{A})$: Cannot decide.
Unsafe due to lack of information.

constraining information determines bounds

You are given one pound of playdoh,
how much mass can you put above a
while keeping the seesaw balanced around m?



$$\begin{cases} \max p \\ \text{subject to } a\,p \le m \end{cases}$$

Answer $\dfrac{m}{a}$

generalizes to bounds on unknown distribution

What is the least upper bound on $\mathbb{P}[X \geq a]$ if all that you know is that $\mathbb{P}$ is an unknown distribution on $[0, 1]$ having mean less than m

$$\mathcal{A} = \{\mu \in \mathcal{M}([0,1]) \,|\, \mathbb{E}_\mu[X] \leq m\}$$

Markov's inequality

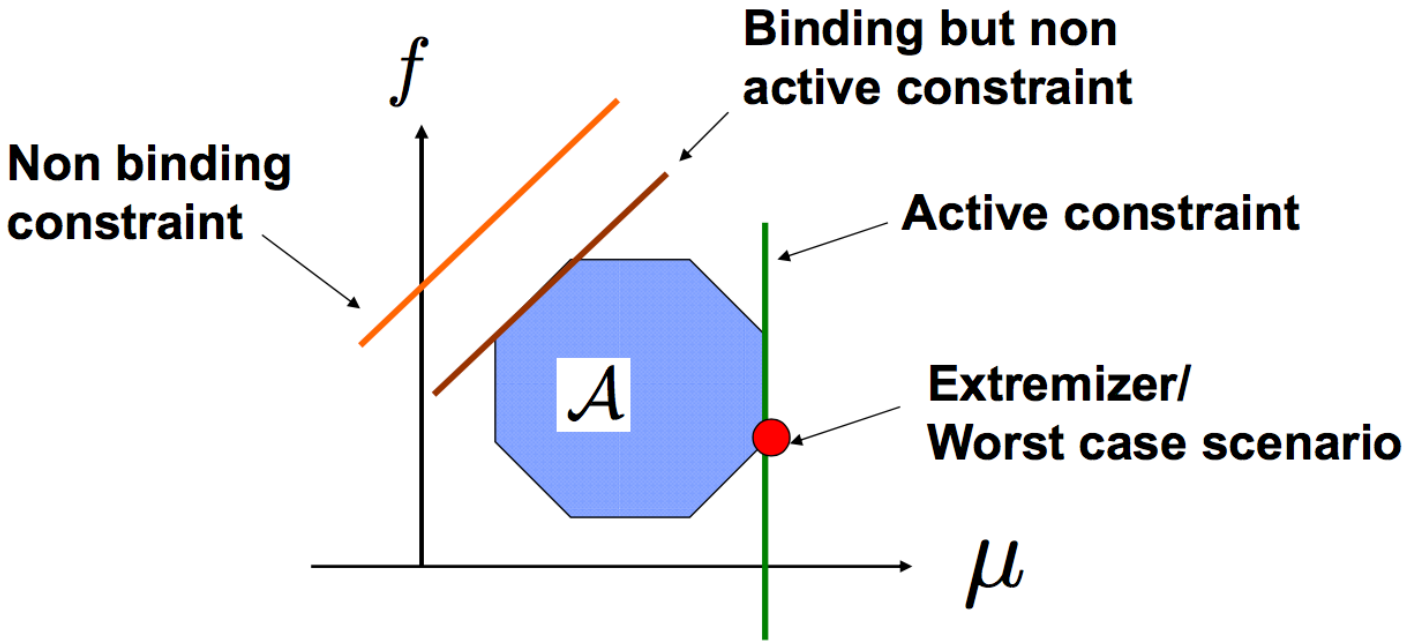
Answer

$$\sup_{\mu \in \mathcal{A}} \mu\left[X \geq a\right] = \frac{m}{a}$$

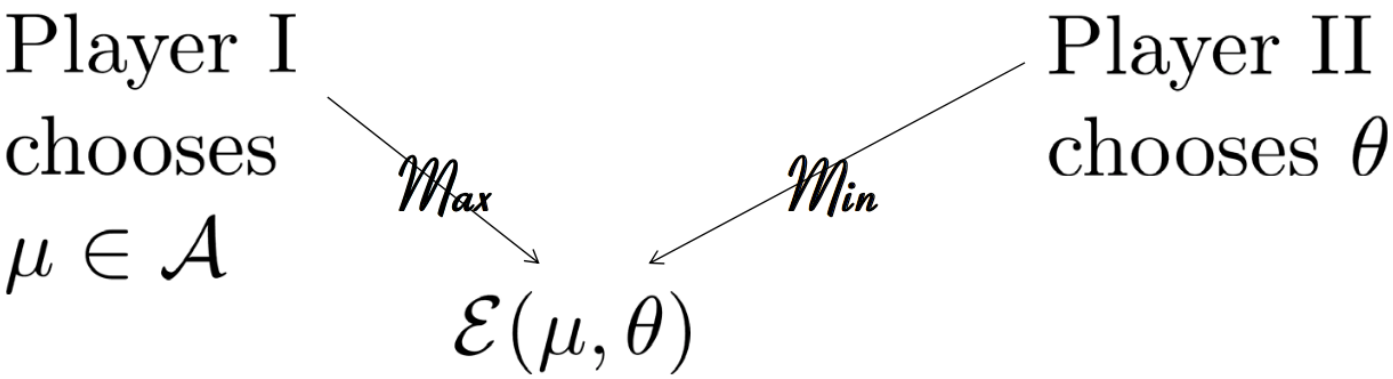
constraining information determines the bounds

Each piece of information is a constraint on an optimization problem.

Optimization concepts (binding, active) transfer to UQ concepts



bounds on expected error as an information game



Pure strategy solution for Player II

Optimal bound on the statistical error

$$\max_{\mu \in \mathcal{A}} \mathcal{E}(\mu, \theta)$$

Optimal statistical estimators

$$\min_{\theta} \max_{\mu \in \mathcal{A}} \mathcal{E}(\mu, \theta)$$

Not a saddle point: $\min_{\theta} \max_{\mu \in \mathcal{A}} \mathcal{E}(\mu, \theta) \neq \max_{\mu \in \mathcal{A}} \min_{\theta} \mathcal{E}(\mu, \theta) = 0$

solving for bounds on statistical quantities

- A rigorous notion of optimality can be derived from the worst case bounds on expected distances of model predictions $\Phi(f,\mu)$ from new data d to be sampled from the unknown data generating distribution $\mathcal{D}(f,\mu)$, which depends on the unknown probability measure μ and response function f drawn from the admissible set $\mathcal{A}$ of potential solutions.
- The goal is to find a function of the data $\theta(d)$ that minimizes the worst case statistical error between the model predictions and $\theta(d)$. If we select an arbitrary (not necessarily optimal) function of the data $\theta^*(d)$, then statistical error is defined by maximizing the distance between $\Phi(f,\mu)$ and $\theta^*(d)$ over the space defined by all $(f,\mu) \in \mathcal{A}$. **The most robust model is the model that minimizes the worst case statistical error over all potential functions of samplings of the data from $\mathcal{D}(f,\mu)$** -- (Owhadi et al, 2015).

optimal model

$$\min_{\theta} \max_{(f,\mu)\in\mathcal{A}} \mathbb{E}_{d\sim\mathbb{D}(f,\mu)}\left[|\theta(d)-\Phi(f,\mu))|^2\right]$$

optimal bounds on the statistical error for a given model

$$\max_{(f,\mu)\in\mathcal{A}} \mathbb{E}_{d\sim\mathbb{D}(f,\mu)}\left[|\theta^*(d)-\Phi(f,\mu))|^2\right]$$

optimal bounds on model uncertainty for a given model

$$\max_{(f,\mu)\in\mathcal{A}} \mathbb{P}_{d\sim\mathbb{D}(f,\mu)}\left[|\theta^*(d)-\Phi(f,\mu))| \geq a\right]$$

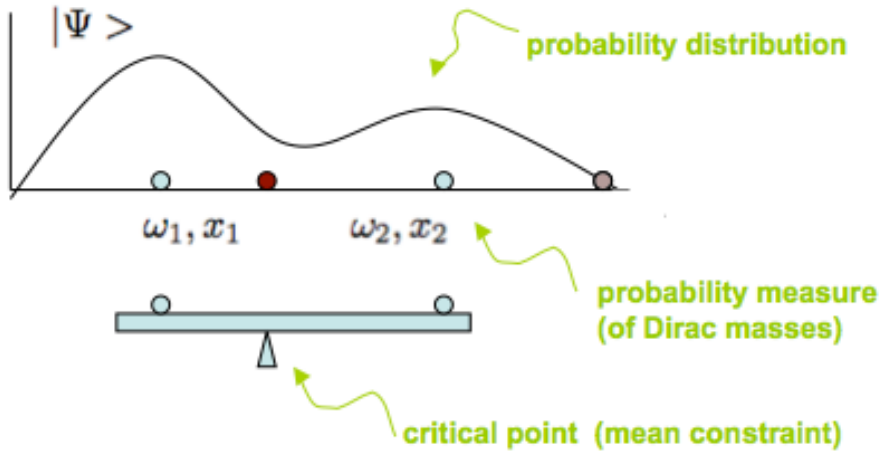
optimal bounds on likelihood of failure for a given model

$$\max_{(f,\mu)\in\mathcal{A}} \mathbb{P}_{d\sim\mathbb{D}(f,\mu)}\left[|\theta^*(d)-\Phi(f,\mu))| \geq a\right] \leq \varepsilon$$

statistical kernels transform to probability space

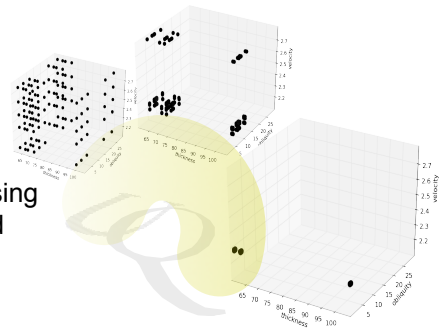
- optimization in product measure space (not input parameter space)

$$|\Psi' > = \hat{c}|\Psi > = \sum_i \omega_i |x_i >$$

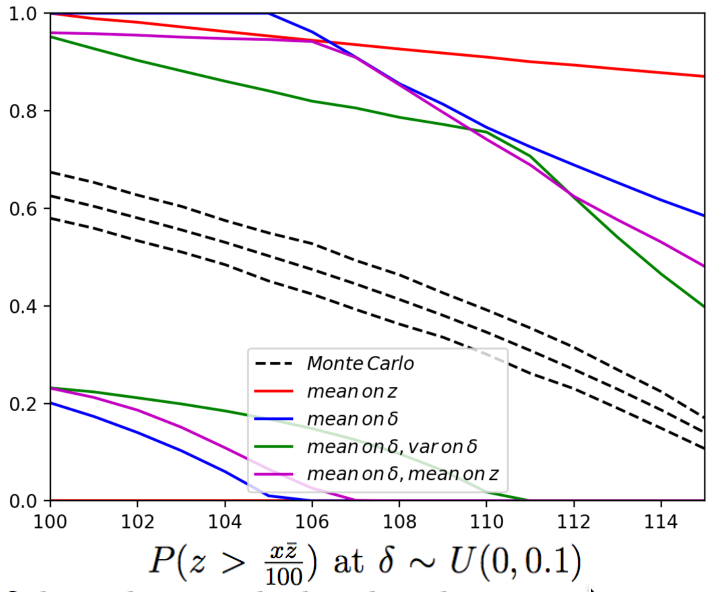


- mean-constrained optimization balances weights and positions of Dirac masses around a critical point

by transforming to product measure space, we maximize the quantity of interest by using optimizers to search over a discretized probability distribution



In measure space, extremum-seeking algorithms seek rare events and discover worst-case bounds, hence generally outperform Monte Carlo sampling.



UQ calculation of bounds on likelihood of failure vs percent distance the next shockfront forms beyond the average of 1M shocks. Notice how UQ bounds respond to each new piece of information while Monte Carlo bounds do not (McKerns et al, 2019).

example: mean constraints in measure space

```
def constraints(rv):
    c = product_measure().load(rv, npts)
    # impose norm on each discrete measure
    for measure in c:
        if not almostEqual(float(measure.mass), 1.0):
            measure.normalize()
    # impose expectation value and other constraints on product measure
    E = float(c.expect(model))
    if E > (target[0] + error[0]) or E < (target[0] - error[0]):
        c.set_expect((target[0],error[0]), model, (x_lb,x_ub), _constraints)
    return c.flatten()  # extract parameter vector of weights and positions

def _constraints(c):
    E = float(c[0].mean)
    if E > (target[1] + error[1]) or E < (target[1] - error[1]):
        c[0].mean = target[1]
    return c

def objective(rv):
    c = product_measure().load(rv, npts)
    return MINMAX * c.pof(failure)
```

$$\mathcal{A} = \left\{ (g,\mu) \;\middle|\; \begin{array}{c} g = \texttt{model} : \mu \in [\texttt{lb},\texttt{ub}] \to \mathbb{R}, \\ \mu = \sum_{i=0}^{3} w_i \delta_i, \\ \sum_{i=0}^{3} w_i = 1, \\ \mathbb{E}_\mu[g] = \texttt{z\_mean}, \\ \bar{\mu} = \texttt{d\_mean} \end{array} \right\}$$

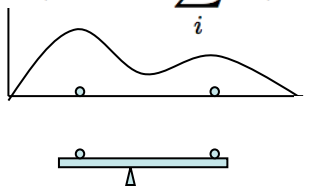
Placing information constraints in kernels (e.g. not built into the model) enables testing how new measurements and information (i.e. adding a new constraint on the inputs or outputs) alters the bounds on all possible outcomes.

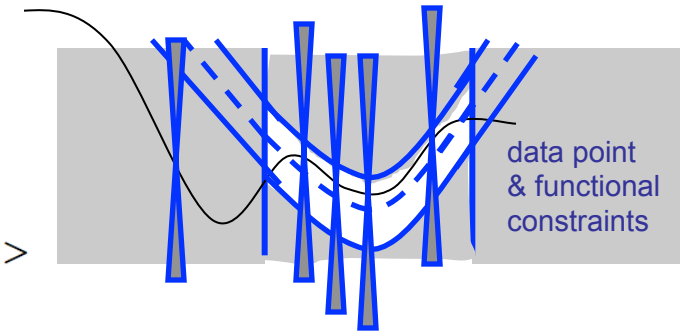
We can perform **design of experiments** to discover an information set that can certify the system (to pass a statistical test within a given tolerance)

example: constraints from data & approx models

```
# generate primary constraints function
def constraints(rv):
  c = scenario()
  c.load(rv, npts)

  # ensure norm(wi) = 1.0 in each discrete measure
  norm = 1.0
  for i in range(len(c)):
    w = c[i].weights
    w[-1] = norm - sum(w[:-1])
    c[i].weights = w

  # impose mean on the values of the product measure
  from mystic.math.discrete import mean_y_norm_wts_constraintsFactory as factory
  constrain = factory((target[0],error[0]), npts)

  # check mean value, and if necessary use constrain to set mean value
  y = float(mean(c.values, c.weights))
  if not (y >= float(target[0] - error[0])):
    c.update( constrain( c.flatten(all=True) ) )

  # then test if valid... then impose model validity on product measure
  if not c.valid_wrt_model(model, ytol=target[2], xtol=target[3], \
                                                  imax=target[4]):
    c.set_valid(model, cutoff=target[2], bounds=bounds, tol=error[2], \
                               constraints=constrain, xtol=target[3], \
                               maxiter=error[3], imax=error[4])
  # extract weights and positions and values
  return c.flatten(all=True)
```

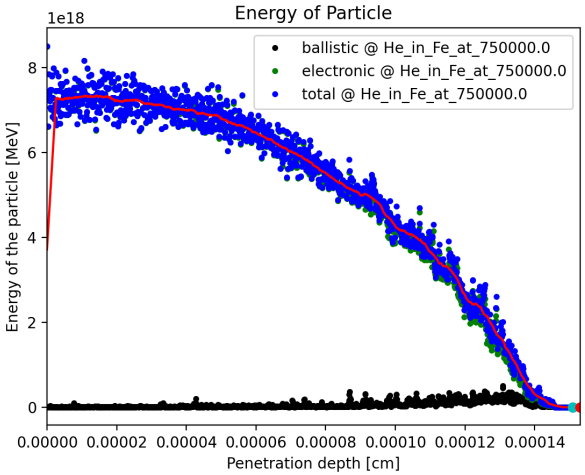
$$|\Psi'> = \hat{c}|\Psi> = \sum_i \omega_i |x_i>$$

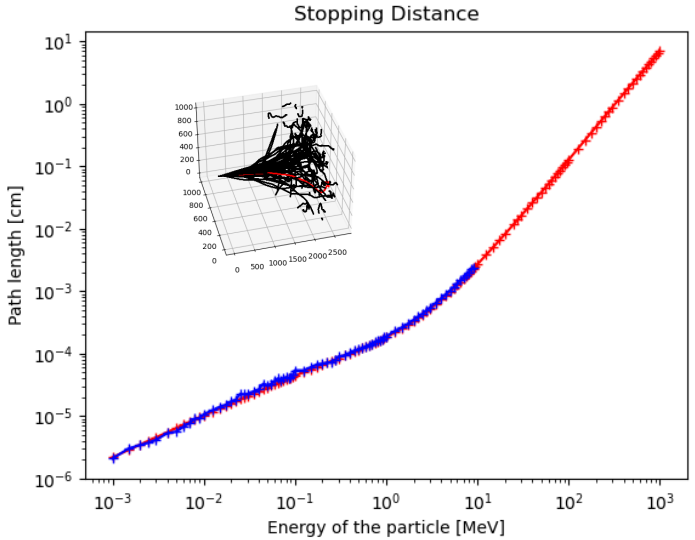




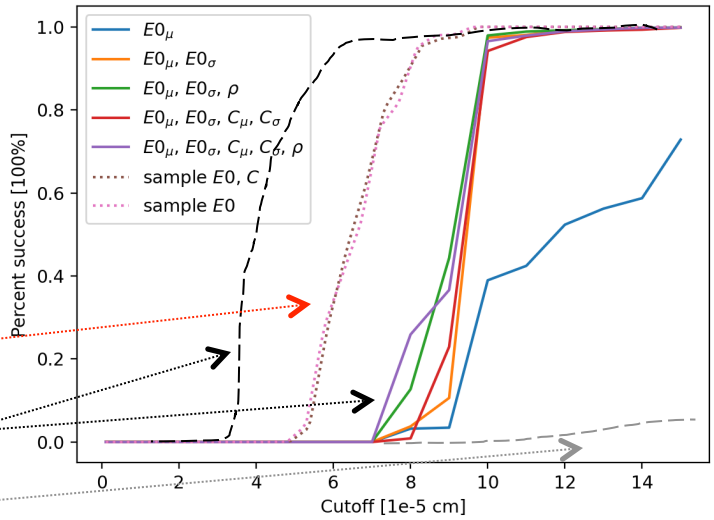
failure of shielding under particle radiation

- simulate He-ion into Fe shielding
  - use Monte Carlo sampling to calculate average penetration depth into shieling





- calculate likelihood of failure
  - failure if particle breaches shielding
- want a design measure (risk)
  - expected penetration depth
  - bound on expected penetration depth
  - bound on worst-case penetration



likelihood of non-elastic failure in tower joint

- Problem: Can we certify the seismic safety of a given structure subjected to earthquake ground motion, where only the maximum magnitude and focal distance of the earthquake are known?

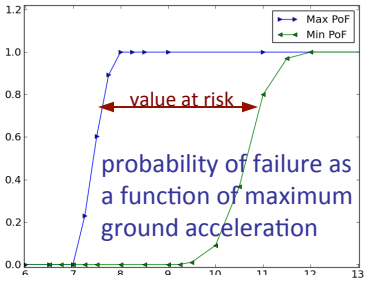
- We construct all possible earthquake scenarios
  - Random inputs of high-dimensionality (~600) with a large number of constraints (~1200)
  - Inputs are coefficients $\boldsymbol{c_i}$ in the transfer function, and amplitudes $\boldsymbol{X_i}$ and durations $\boldsymbol{s_i}$ in the earthquake source function

$$s(t) := \sum_{i=1}^{B} X_i \, s_i(t) \qquad \psi(t) := \frac{\sqrt{q}}{\tau'} \sum_{i=1}^{q} c_i \, \varphi_i(t)$$

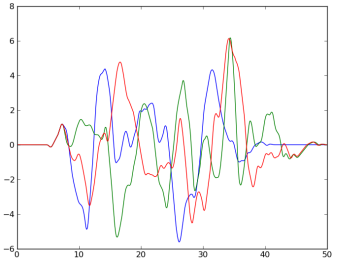
- Ground acceleration is a convolution of the source and transfer functions, while dynamics of joint deflection are governed by

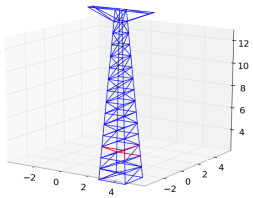
$$v_\alpha(t) = -\int_0^t \mathrm{e}^{-\zeta_\alpha \omega_\alpha (t-\tau)} \sin[\omega_\alpha (t-\tau)] \left(q_\alpha^T M T \ddot{u}_0(\tau)\right) \frac{\mathrm{d}\tau}{\omega_\alpha}$$

$$\ddot{u}_0(t) := (\psi \star s)(t)$$

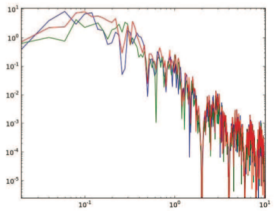


typical scenarios for resulting ground acceleration





a truss structure



when axial strain occurs near truss resonance modes, failure can occur

- Failure occurs when axial strain in any truss member exceeds the member yield strain

$$\|L_i v\|_\infty < S_i$$

- We determine the probability of non-elastic failure with respect to the unknown earthquake ground motion the structure will experience

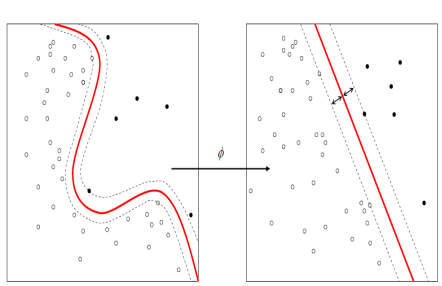
Owhadi et al, SIAM Review 2012

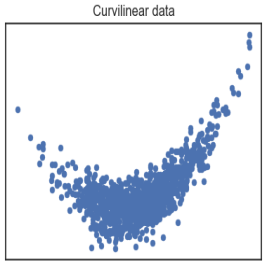
can we better utilize physical information?

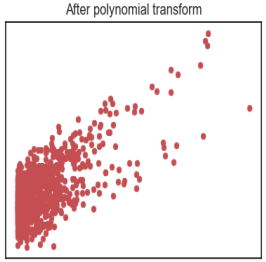
- kernel transforms often are used to incorporate nonlinear information into linear models

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D$$

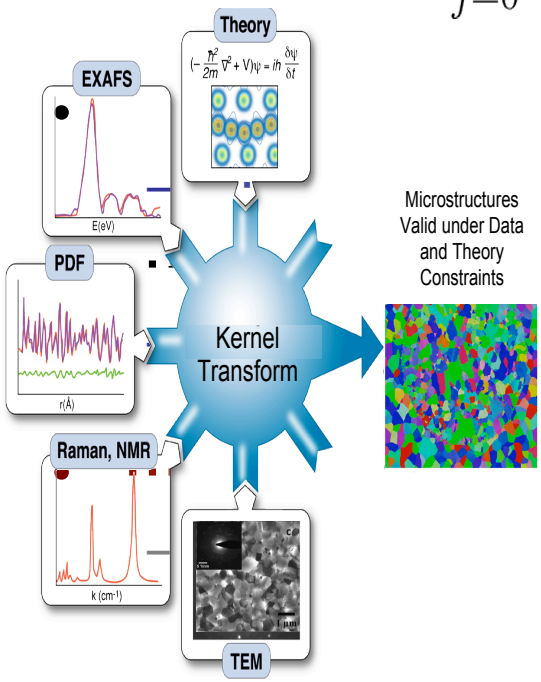
$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x})$$







*A kernel transform can be seen to transform the model **f'(x)** = **f(c(x))** or to transform the input coordinates **f(x')** = **f(c(x))***



Constraining information from measurements and theory are used to construct a kernel. Learning is then performed in the space of valid solutions

Can we build kernels that ensure models are trained in a space that guarantees they are valid with respect to all known physical and statistical constraints?

If so, can we think of hierarchical learning in terms of hierarchical kernel transformations?

example: design optimization w/ soft constraints

```
"Pressure Vessel Design"

from vessel import objective, bounds, xs, ys

from mystic.constraints import as_constraint
from mystic.penalty import quadratic_inequality

def penalty1(x): # <= 0.0
    return -x[0] + 0.0193*x[2]

def penalty2(x): # <= 0.0
    return -x[1] + 0.00954*x[2]

def penalty3(x): # <= 0.0
    from math import pi
    return -pi*x[2]**2*x[3] - (4/3.)*pi*x[2]**3 + 1296000.0

def penalty4(x): # <= 0.0
    return x[3] - 240.0

@quadratic_inequality(penalty1, k=1e12)
@quadratic_inequality(penalty2, k=1e12)
@quadratic_inequality(penalty3, k=1e12)
@quadratic_inequality(penalty4, k=1e12)
def penalty(x):
    return 0.0

solver = as_constraint(penalty)
```

```
if __name__ == '__main__':

    from mystic.solvers import diffev2
    from mystic.math import almostEqual

    result = diffev2(objective, x0=bounds, \
                     bounds=bounds, \
                     penalty=penalty, \
                     npop=40, gtol=500)
```

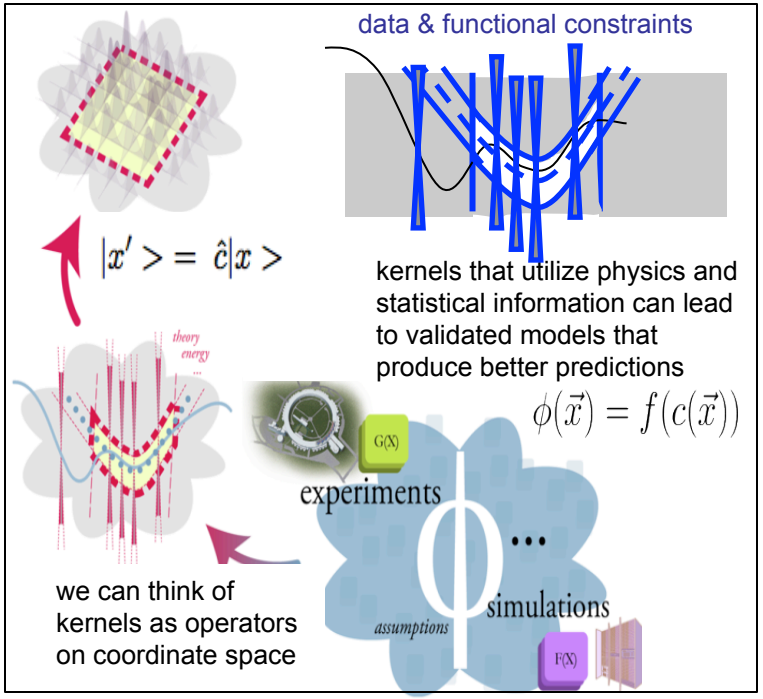
example: global MIP w/ symbolic constraints

```
def objective(x):
    return 0.0

bounds = [(0,10)]*7
# constraints
equations = """
98527*x0 + 34588*x1 + 5872*x2 + 59422*x4 + 65159*x6 - 1547604 - 30704*x3 - 29649*x5 == 0.0
98957*x1 + 83634*x2 + 69966*x3 + 62038*x4 + 37164*x5 + 85413*x6 - 1823553 - 93989*x0 == 0.0
900032 + 10949*x0 + 77761*x1 + 67052*x4 - 80197*x2 - 61944*x3 - 92964*x5 - 44550*x6 == 0.0
73947*x0 + 84391*x2 + 81310*x4 - 1164380 - 96253*x1 - 44247*x3 - 70582*x5 - 33054*x6 == 0.0
13057*x2 + 42253*x3 + 77527*x4 + 96552*x6 - 1185471 - 60152*x0 - 21103*x1 - 97932*x5 == 0.0
1394152 + 66920*x0 + 55679*x3 - 64234*x1 - 65337*x2 - 45581*x4 - 67707*x5 - 98038*x6 == 0.0
68550*x0 + 27886*x1 + 31716*x2 + 73597*x3 + 38835*x6 - 279091 - 88963*x4 - 76391*x5 == 0.0
76132*x1 + 71860*x2 + 22770*x3 + 68211*x4 + 78587*x5 - 480923 - 48224*x0 - 82817*x6 == 0.0
519878 + 94198*x1 + 87234*x2 + 37498*x3 - 71583*x0 - 25728*x4 - 25495*x5 - 70023*x6 == 0.0
361921 + 78693*x0 + 38592*x4 + 38478*x5 - 94129*x1 - 43188*x2 - 82528*x3 - 69025*x6 == 0.0
"""

from mystic.symbolic import generate_penalty, generate_conditions
pf = generate_penalty(generate_conditions(equations))

from numpy import round as nround


if __name__ == '__main__':

    from mystic.solvers import diffev2
    result = diffev2(objective, x0=bounds, bounds=bounds, penalty=pf,
                     constraints=nround, npop=40, gtol=50, disp=True, full_output=True)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 88
         Function evaluations: 3560
[ 6.  0.  8.  4.  9.  3.  9.]
```

physics-informed kernels increase model validity

- box (range) constraints
- nonlinear (functional) constraints
- uniqueness and set-membership constraints
- probabilistic and statistical constraints
- constraints imposing sampling statistics
- inputs from sampling distributions
- constraints from legacy data (points and data sets)
- constraints from models and distance metrics
- constraints on (product) measures
- support vector (weight, independence) collapse




```
>>> from mystic.constraints import unique, discrete, integers, with_mean, and_, not_
>>> from mystic.math.measures import mean
>>> c = and_(unique, discrete(range(10,100,3))(lambda x: x), with_mean(50)(lambda x:x))
>>> c([6,33,14,33,51])
[89.0, 44.0, 50.0, 32.0, 35.0]
>>> mean(_)
50.0
>>> c = and_(integers()(lambda x:x), not_(lambda x:[0]*len(x)), with_mean(0)(lambda x:x))
>>> c([6,3,-1,-3,5])
[4, 1, -3, -5, 3]
>>> mean(_)
0.0
```

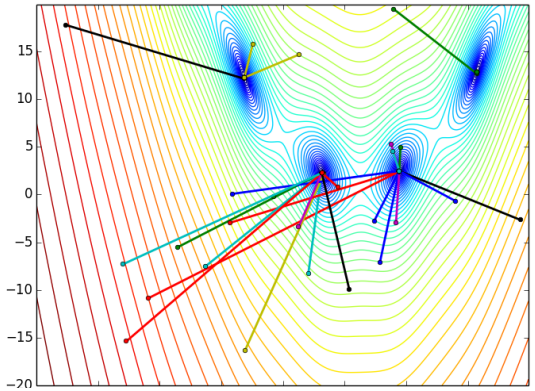
instead of training purely on data, models are fit in a space defined by physical and statistical constraints -- thus are guaranteed to be valid

example: information-constrained learning

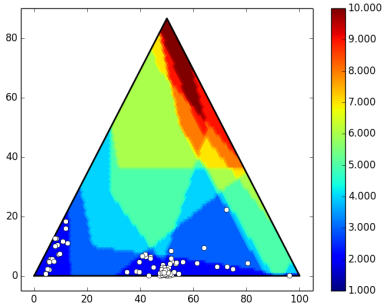
```
>>> import numpy as np
>>> from sklearn import preprocessing as pre
>>> from sklearn import linear_model as lin
>>> from mystic.symbolic import generate_constraint, generate_solvers, simplify
>>> from mystic.constraints import vectorize
>>>
>>> # define a model
>>> a,b,c,d = 0.661, -1.234, 2.983, -16.5571
>>> def model(x):
...   x0,x1,x2,x3 = x
...   return a*x3**3 + b*x2**2 + c*x1 + d*x0
...
>>> # generate some sparse data
>>> xtrain = np.random.uniform(0,100, size=(10,4))
>>> target = model(xtrain.T).T
>>> xtest = np.random.uniform(0,100, size=(10,4))
>>> test = model(xtest.T).T
>>>
>>> # define some model constraints
>>> equations = """
... 3*b + c > -0.75
... 4.5*b - d > 11.0
... """
>>> var = list('abcd')
>>> equations = simplify(equations, variables=var)
>>> cf = generate_constraint(generate_solvers(equations, variables=var))
>>>
>>> # define a kernel-transformed regressor
>>> ta = pre.FunctionTransformer(func=vectorize(cf, axis=1))
>>> tp = pre.PolynomialFeatures(degree=3)
>>> e = lin.LinearRegression()
>>>
>>> # train and score, then test and score
>>> xtrain_ = tp.fit_transform(ta.fit_transform(xtrain))
>>> e.fit(xtrain_, target).score(xtrain_, target)
1.0
>>> xtest_ = tp.fit_transform(ta.fit_transform(xtest))
>>> e.score(xtest_, test)
0.9999932741261055
>>>
```

can we perform these calculations efficiently?

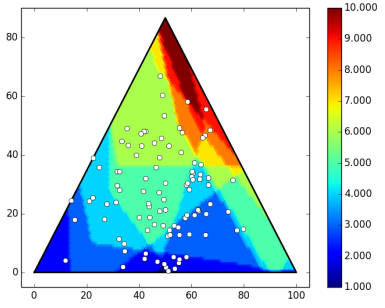
- asynchronous parallel ensemble optimization provides orders-of-magnitude speedup for multi-layer and global optimizations



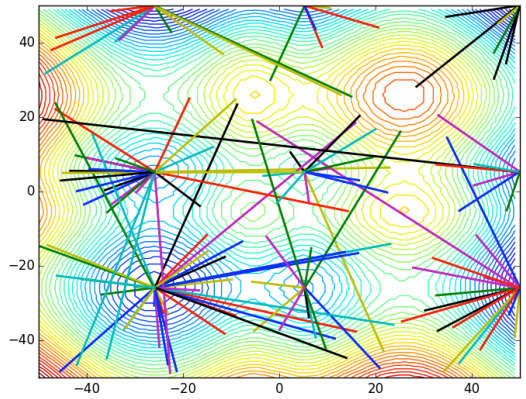
Single Buckshot Powell search for all minima



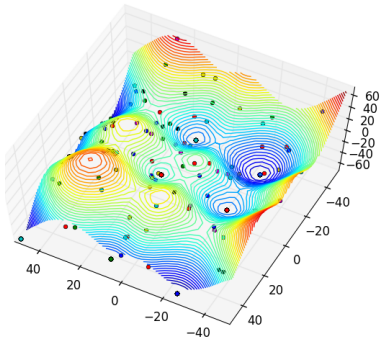
Diff Ev: 9500s (100 points at 95s /point) population of 40



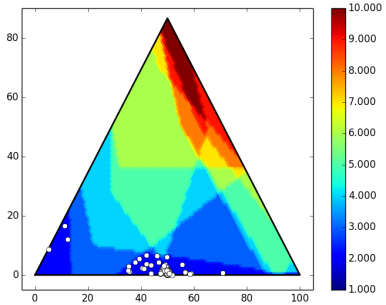
Simplex: 1000s (100 points at 10s /point)



Multi-iteration Buckshot Powell search for all minima.



a parallel ensemble of simplex solvers performs better than a genetic algorithm, and in much less time.



Buckshot Simplex: 200s for batch of 100 solvers on 512 cores

```
dude@hilbert>$ python global_search.py
CacheInfo(hit=17, miss=8, load=0, maxsize=None, size=8)
CacheInfo(hit=24, miss=1, load=0, maxsize=None, size=9)
CacheInfo(hit=25, miss=0, load=0, maxsize=None, size=9)
CacheInfo(hit=25, miss=0, load=0, maxsize=None, size=9)
min: -70.8861291838 (count=1)
pts: 9 (values=8, size=9)
```

"cache" is an abstraction on storage. "load" is local memory cache, while "hit" is an archive hit. "miss" is a new point. Results shown are for when configured for direct connectivity with archival database.

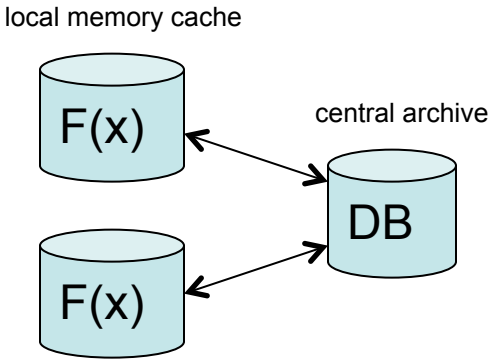
parallel graph execution and statefulness

PATHOS
a framework for parallel graph management and execution in heterogeneous computing

```
# the function to be minimized and the bounds
from mystic.models import rosen as my_model
lb = [0.0, 0.0, 0.0]; ub = [2.0, 2.0, 2.0]

# get monitor and termination condition objects
from mystic.monitors import LoggingMonitor
stepmon = LoggingMonitor(1, 'log.txt')
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# select the parallel launch configuration
from pyina.launchers import TorqueMpi
my_map = TorqueMpi('25:ppn=8').map

# instantiate and configure the nested solver
from mystic.solvers import PowellDirectionalSolver
my_solver = PowellDirectionalSolver(len(lb))
my_solver.SetStrictRanges(lb, ub)
my_solver.SetEvaluationLimits(1000)

# instantiate and configure the outer solver
from mystic.solvers import BuckshotSolver
solver = BuckshotSolver(len(lb), 200)
solver.SetRandomInitialPoints(lb, ub)
solver.SetGenerationMonitor(stepmon)
solver.SetNestedSolver(my_solver)
solver.SetSolverMap(my_map)
solver.Solve(my_model, COG)
# obtain the solution
solution = solver.bestSolution
```

- available launchers:
  - multiprocess, threaded
  - MPI parallel
  - RPC/IPC (distributed)
  - SSH
- available schedulers:
  - torque, slurm, lsf

```
# couple the LAMMPS wrapper to an archive for LAMMPS data
@mystic.cache.cached(archive='Data_DB', multivalued=True)
def model(x, axis=None):
    if axis is None: axis = slice(None)
    # convert the 1-D input array to a tuple
    # of arguments for the LAMMPS function
    return LAMMPS(*convert(x))[axis]
```



automated state saving and sharing

cache-to-archive interaction

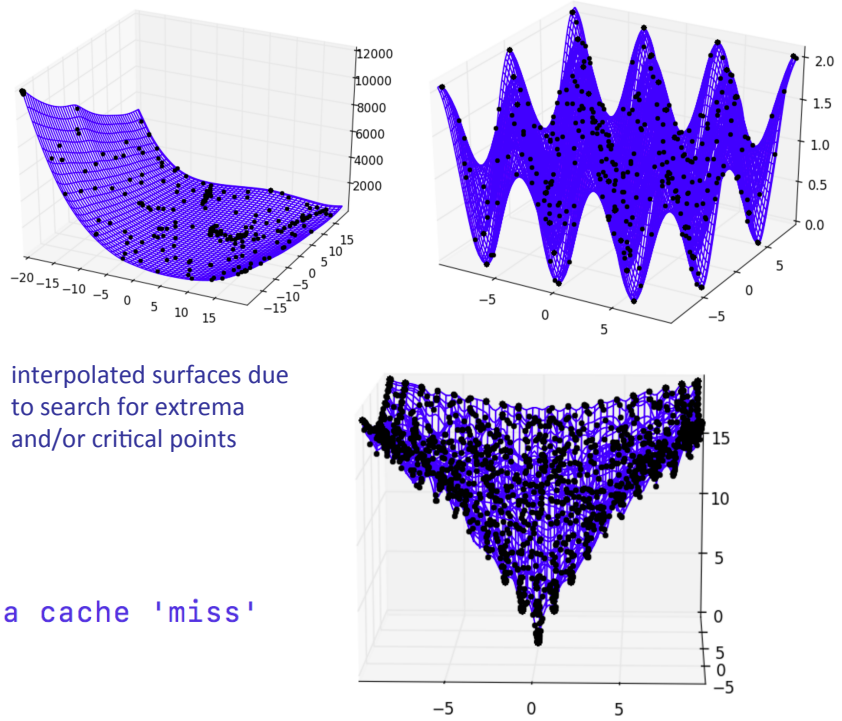
caching to memory, hdf, file, directory, database

asynchronous execution enables active learning

```
>>> import mystic as my
>>> import mystic.models as mm
>>> solver = my.solvers.DifferentialEvolutionSolver(4,40)
>>> solver.SetObjective(mm.rosen)
>>> solver.SetInitialPoints([10,9,8,7])
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
92529.70241985259
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
650.8598624082304
>>> solver.bestSolution
array([ 0.16878542, -0.86436478,  2.81666842,  9.09712308])
>>> constraint = my.constraints.integers()(lambda x:x)
>>> solver.SetConstraints(constraint)
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
509.0
>>> solver.bestSolution
array([ 0., -1.,  3.,  9.])
>>> solver.Step()
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
205.0
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
205.0
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
205.0
>>> solver.SetConstraints(None)
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
53.78841867669052
>>> solver.bestSolution
array([1.        , 1.67260416, 3.        , 9.        ])
>>> solver.Solve()
>>> solver.Terminated(info=True)
"VTRChangeOverGeneration with {'gtol': 1e-06, 'target': 0.0, 'generations': 30, 'ftol': 0.005}"
>>> solver.bestEnergy
53.788374768170485
>>> stop = my.termination.VTRChangeOverGeneration(generations=60)
>>> solver.SetTermination(stop)
>>> solver.SetInitialPoints(solver.population[0])
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
3.6000884492638634
>>> solver.bestSolution
array([1.28412131, 1.65897873, 2.75352996, 7.58261074])
>>> solver.Step()
>>> solver.Step()
>>> solver.Step()
>>> solver.Step()
>>> solver.bestEnergy
1.2847519993127898
>>> solver.bestSolution
array([1.17162622, 1.38946827, 1.96748706, 3.86501662])
>>> solver.Solve()
>>> solver.bestSolution
array([1.00117567, 1.00352551, 1.00672722, 1.01106216])
>>> solver.Terminated(info=True)
"VTRChangeOverGeneration with {'gtol': 1e-06, 'target': 0.0, 'generations': 60, 'ftol': 0.005}"
>>> stop = my.termination.VTRChangeOverGeneration(generations=120, ftol=1e-6)
>>> solver.SetTermination(stop)
>>> solver.Solve()
>>> solver.bestEnergy
8.928616714985148e-07
>>> solver.bestSolution
array([0.99993566, 0.99989024, 0.99983865, 0.9996083 ])
>>> stop = my.termination.VTRChangeOverGeneration(generations=200, ftol=1e-8)
>>> solver.SetTermination(stop)
>>> solver.SetConstraints(constraint)
>>> solver.Step()
"VTRChangeOverGeneration with {'gtol': 1e-06, 'target': 0.0, 'generations': 200, 'ftol': 1e-08}"
>>> solver.bestEnergy
0.0
>>> solver.bestSolution
array([1., 1., 1., 1.])
```

example: accurate nonlinear interpolation / ML

```
from mystic.search import Searcher
from mystic.termination import VTR, ChangeOverGeneration as COG
from klepto.archives import dir_archive
from pathos.pools import ProcessPool as Pool

# cost function
from mystic.models import griewangk as model
ndim = 2  # model dimensionality
bounds = ndim * [(-9.5,9.5)] # griewangk

# the solvers
from mystic.solvers import SparsitySolver
from mystic.solvers import PowellDirectionalSolver
sprayer = SparsitySolver
seeker = PowellDirectionalSolver
npts = 25 # number of solvers
stop = COG(1e-4)
_map = Pool().map
retry = 1 # max consectutive iteration retries without a cache 'miss'
tol = 8   # rounding precision
mem = 1   # cache rounding precision

stepmon = None
archive = None

searcher = Searcher(npts, retry, tol, mem, _map, archive, sprayer, seeker)
searcher.Reset(archive, inv=False)
searcher.Search(model, bounds, stop=stop, monitor=stepmon)
searcher._summarize()

##### extract results #####
xyz = searcher.Samples()
```

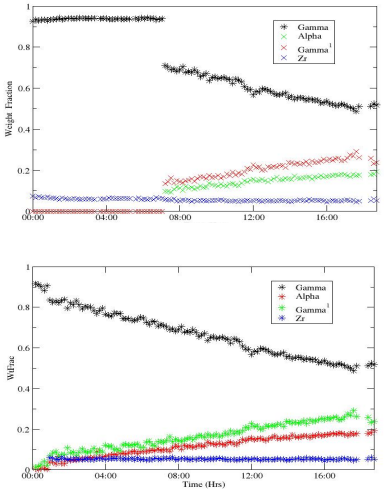


interpolated surfaces due to search for extrema and/or critical points

Interpolation or ML on points generated from a global search (for all extrema and/or critical points) can yield much more accurate nonlinear surrogate models than pure sampling methods.

example: neutron diffraction data analysis

- data analysis typically requires an expert to provide a good initial guess that is already very close to ground truth
  - historically, high-dimensional nonlinear optimizations are attempted with fast local optimizers (in Rietveld refinement)
  - it can take an expert analyst days to months to produce a good refinement, especially for parameters with large nonlinear sensitivity



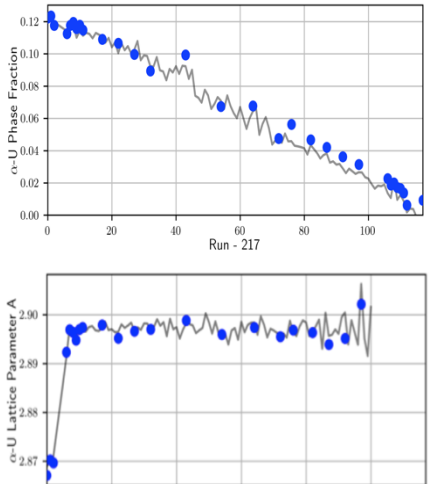
U-Mo study with U-alpha starting at (2.836, 5.867, 4.936)

U-Mo study with U-alpha starting at (2.891, 5.841, 5.015)

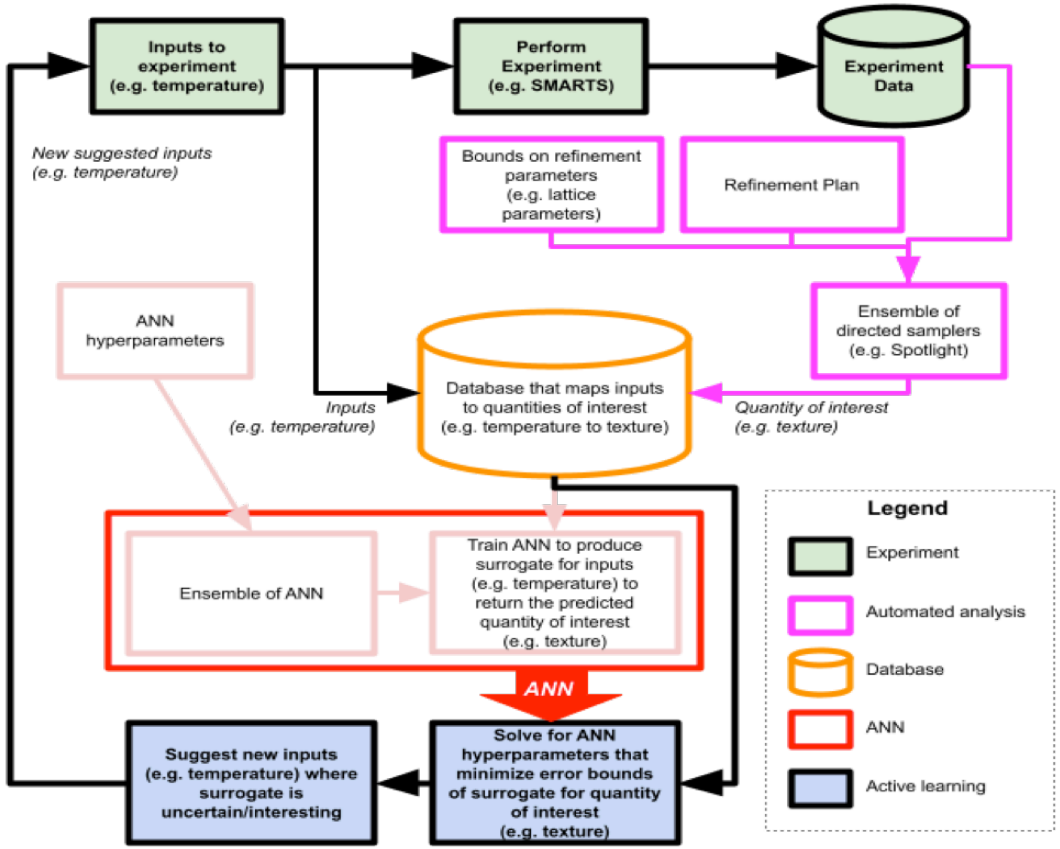
(Δ1.94%, Δ-0.44%, Δ1.60%)

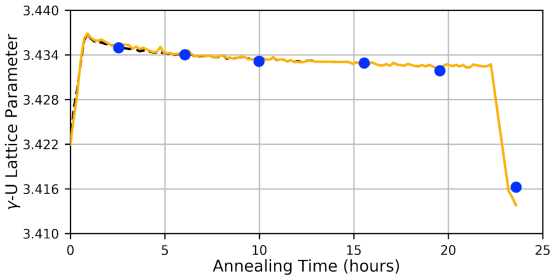
months

a naïve parallel ensemble search to match lattice parameters and weight fraction produced the same results in hours (Biwer et al, 2019).

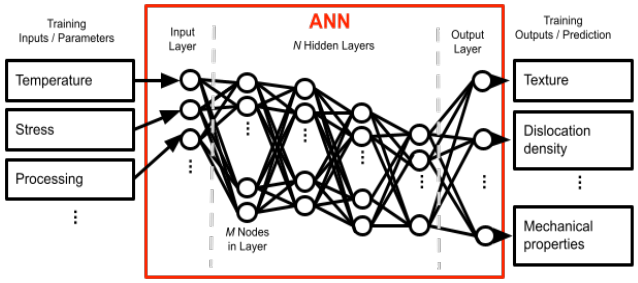


example: robust prediction of materials properties



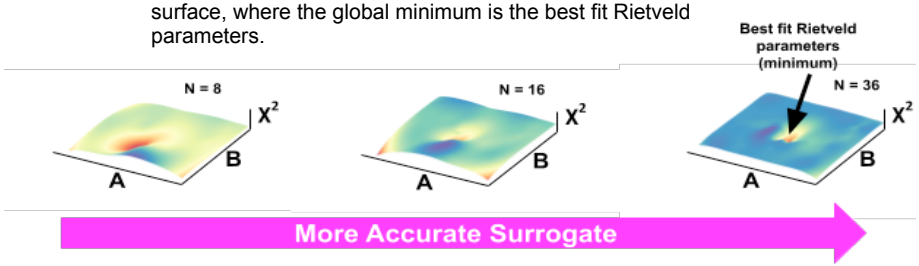


The evolution of the γ-U lattice parameter during annealing of uranium 10%-wt. molybdenum at 490C. The predicted lattice parameter from Spotlight (blue dots) and a scripted refinement using its results (yellow) are overlaid on the results from an expert (dashed black) for comparison.

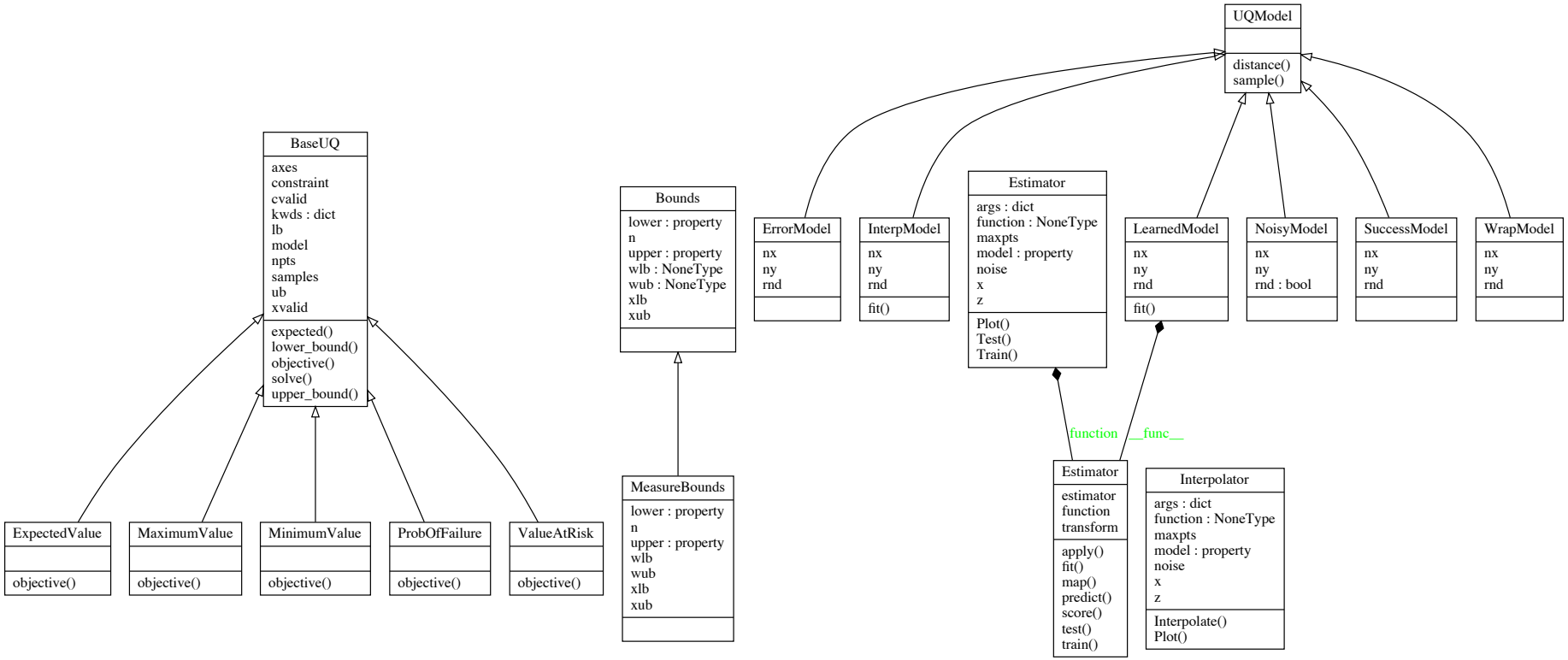


An ensemble of active samplers is used to discover the critical points of two lattice parameters for PbSO4. Results are shown for a single pass of N samplers, where each sampler is driven by a gradient descent optimizer. As N is increased (left to right), the accuracy of a surrogate trained on the sampled data increases. We use sampling to find the critical points on the R-factor (e.g. chi-squared) surface, where the global minimum is the best fit Rietveld parameters.

- in-situ loading/heating measurement (SMARTS)
- Spotlight automates Rietveld refinement (GSAS)
- MILK automates Texture Analysis (MAUD)
- mystic automates active learning with UQ+ML
- uses expected misfit as the quality metric



modular workflow for statistics under uncertainty

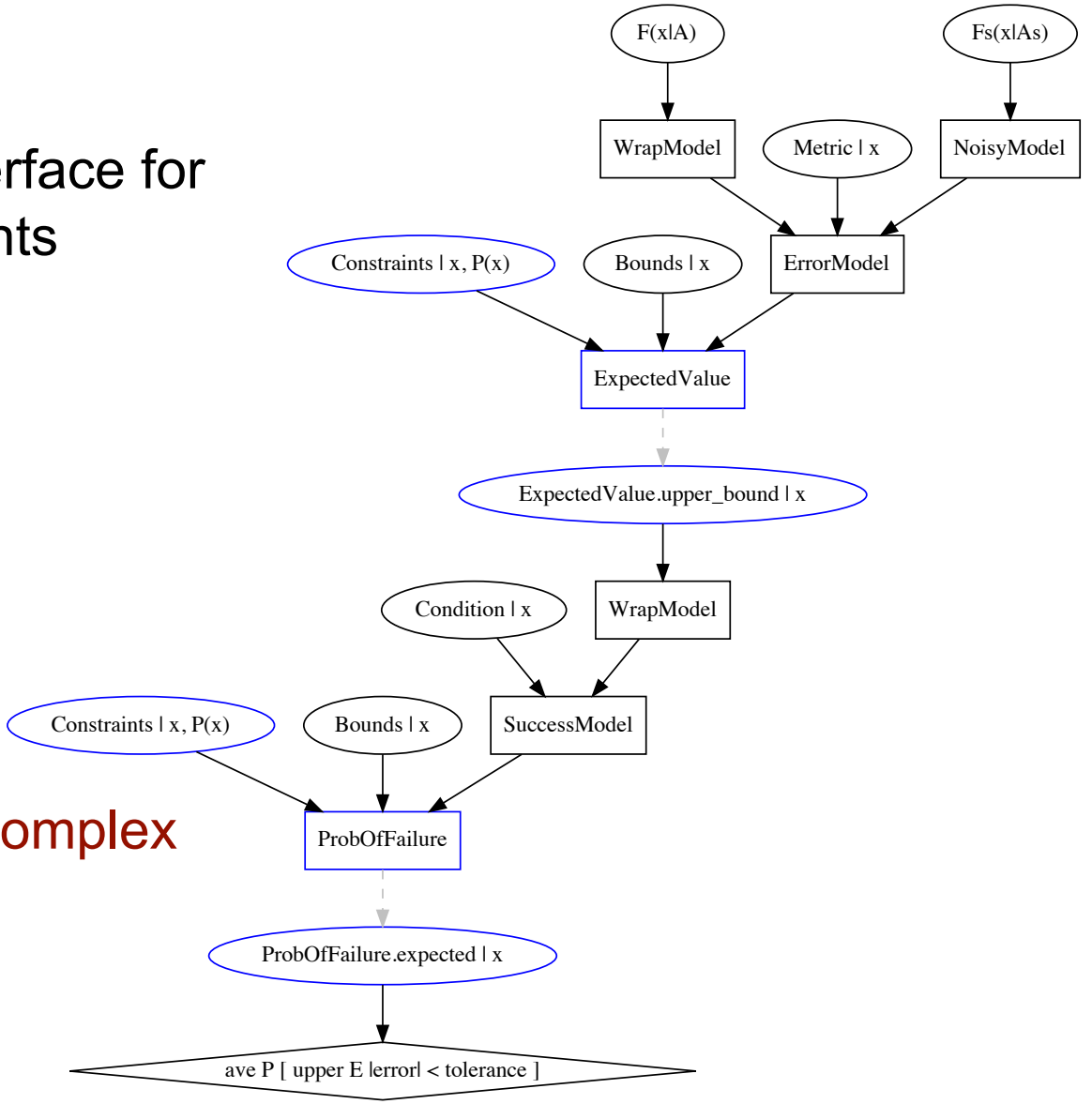


- unified interface for statistical quantities
  - expected value and bounds thereof
  - plug in the UQ approach (OUQ, Bayesian inverse, GP, ...)
- unified interface to models “used in UQ calculations”
  - built-in methods for sampling, connectivity to data archive
  - interpolators & ML estimators for automated surrogate construction

quickly assemble very complex workflows

- simple high-level interface for interacting components
  - "UQ models"
  - statistical quantities
  - bounds
  - constraints
  - penalties
  - conditions (bool)

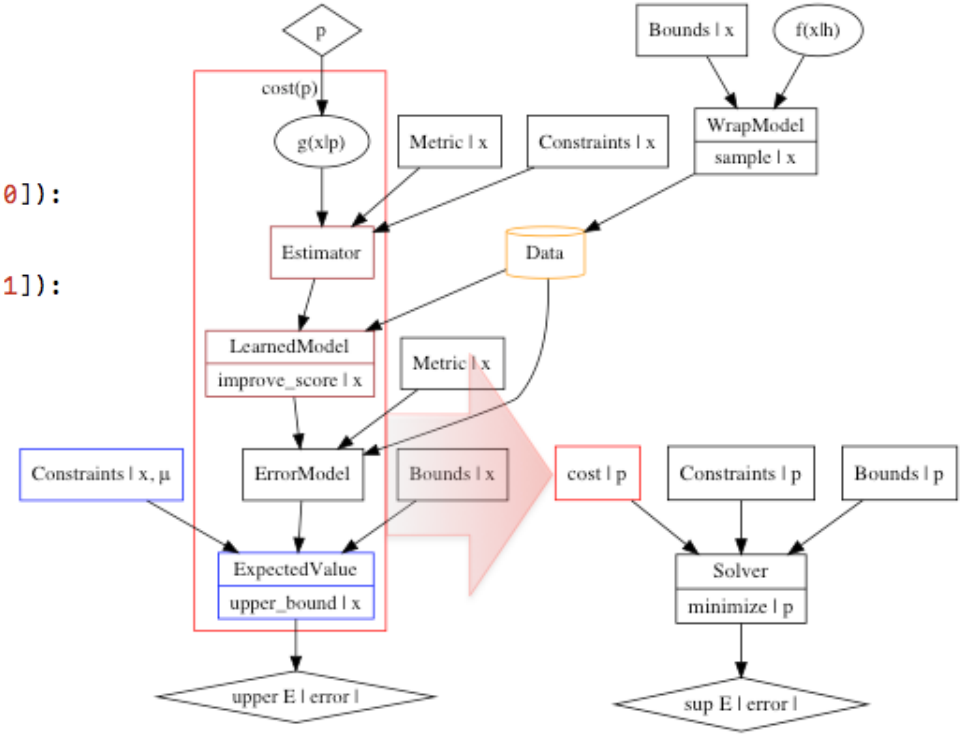
- rapid exploration of complex statistical workflows
  - very few lines of code
  - customizable down to extremely low-level



UQ-driven learning of optimal statistical estimator

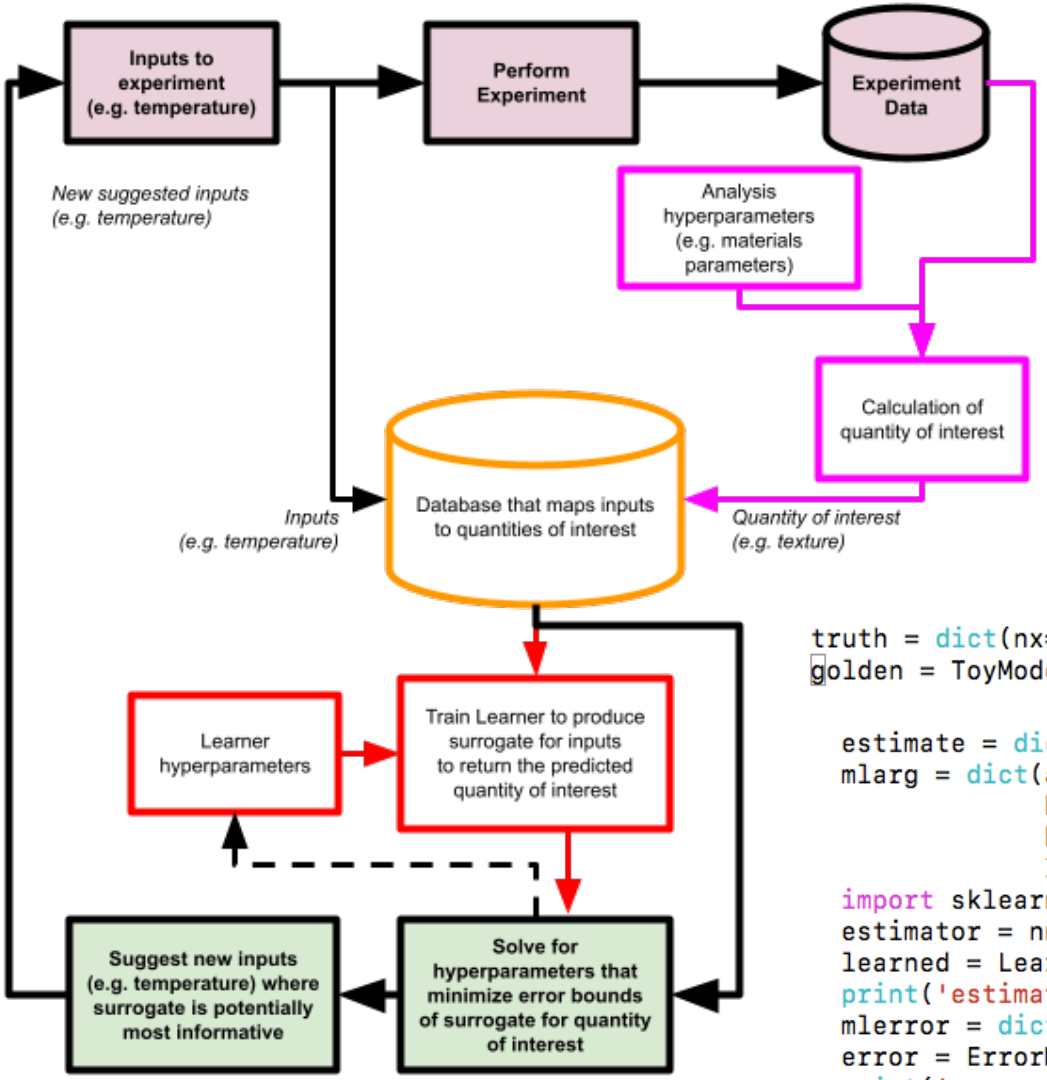
```python
def moment_constraints(c):
    'impose moment constraints on product measure'
    target = T_ave, T_var
    error = T_ave_err, T_var_err
    E = float(c[0].mean)
    if E > (target[0] + error[0]) or E < (target[0] - error[0]):
        c[0].mean = target[0]
    E = float(c[0].var)
    if E > (target[1] + error[1]) or E < (target[1] - error[1]):
        c[0].var = target[1]
    return c
```

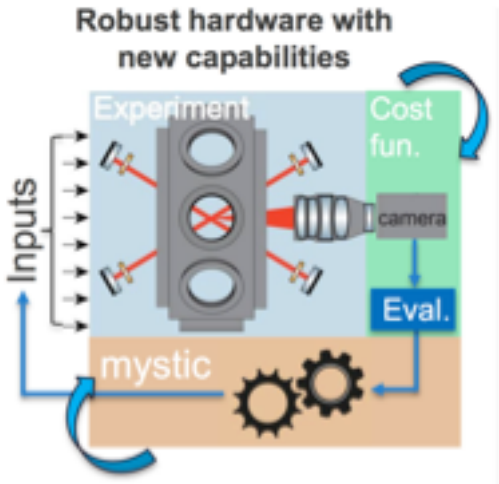
```python
#print("building truth F'(x|a')...")
true = dict(mu=.01, sigma=0., zmu=-.01, zsigma=0.)
truth = ToyModel('truth', nx=5, ny=3, **true)
#print('sampling truth...')
data = truth.sample([(0,1)]+[(0,10)]*4, pts=-16)
```




```python
def cost(x, axis=None):
    # CASE 1: F(x|a) = F'(x|a'). Tune A for optimal G.
    kwds = dict(smooth=x[0], noise=x[1], method='thin_plate', extrap=False)

    #print('building estimator G(x) from truth data...')
    surrogate = InterpModel('surrogate', nx=5, ny=3, data=truth, **kwds)
    #print('building UQ model of model error...')
    error = ErrorModel('error', model=truth, surrogate=surrogate)

    rnd = 25 if error.rnd else None
    #print('building UQ objective of expected model error...')
    b = ExpectedValueOUQ(error, bnd, pmcons=moment_constraints, \
                         iscons=is_constrained, samples=rnd)
    i = counter.count()
    #print('solving for upper bound on expected model error...')
    solver = b.upper_bound(axis=axis, id=i, **param)
```

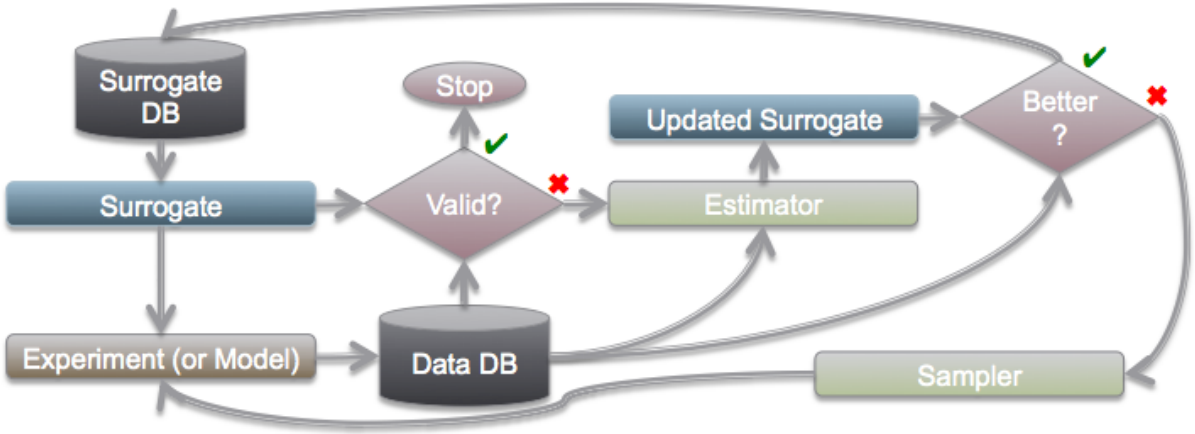
active learning of expected instrument response






```
truth = dict(nx=5, ny=3, mu=.001, zmu=-.001)
golden = ToyModel('golden', cached=True, sigma=0, zsigma=0, **truth)

estimate = dict(nx=5, ny=3, data=golden)
mlarg = dict(alpha=0.0001, batch_size='auto', beta_1=0.9, \
             beta_2=0.999, epsilon=1e-08, \
             hidden_layer_sizes=(100,75,50,25), \
             learning_rate_init=0.001, max_fun=15000)
import sklearn.neural_network as nn
estimator = nn.MLPRegressor(**mlarg)
learned = LearnedModel('learned', estimator=estimator, **estimate)
print('estimate: %s' % str(learned([1,2,3,4,5])))
mlerror = dict(model=golden, surrogate=learned)
error = ErrorModel('error', **mlerror)
print('error: %s' % str(error([1,2,3,4,5])))
```

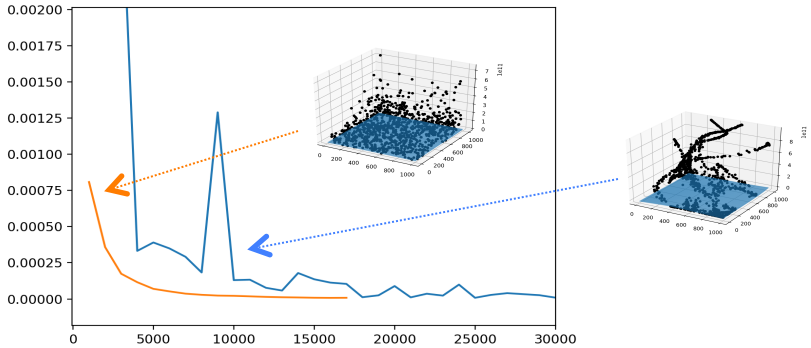
UQ-driven active learning and dynamic sampling



- ensemble sampling:
  - uses an ensemble of local optimizers to discover critical points of the unknown surface
  - can customize the local solver
  - npts is the size of the ensemble
  - sample (asynchronous)
  - sample_until (blocking)
    - evals, iters, terminated

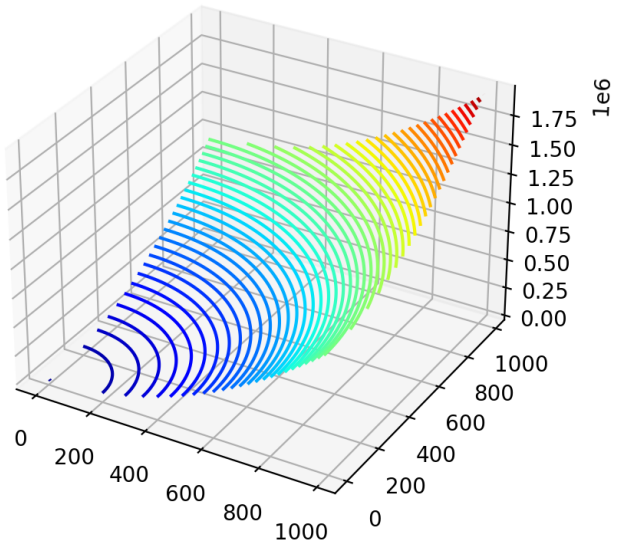
```
import mystic.samplers as sam
import mystic.solvers as sol
s = sam.LatticeSampler(bounds, model, npts=4,
    solver=sol.PowellDirectionalSolver)
s.sample_until(terminated=all)
```

```
# create handles to surrogate archives
# (one surrogate for each one of N outputs)
import mystic.cache.function as func
surr = lambda i: func.db('surrogate{i}.db'.format(i=i))
archives = list(map(surr, range(N)))

# read the surrogates from the archives
# and combine to a single surrogate
surrogate = func.read(archives)
```



test score (on newly encountered data) converges over time

UQ-driven active learning and dynamic sampling

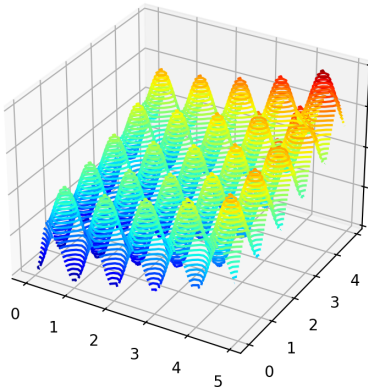


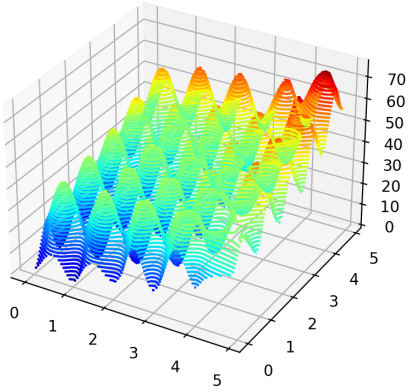
At a cursory level, both traditional sampling from a distribution and optimizer-driven sampling seem to reproduce the Rastrigin function.

However, note that upon zooming in the Rastrigin function has numerous local minima and maxima.

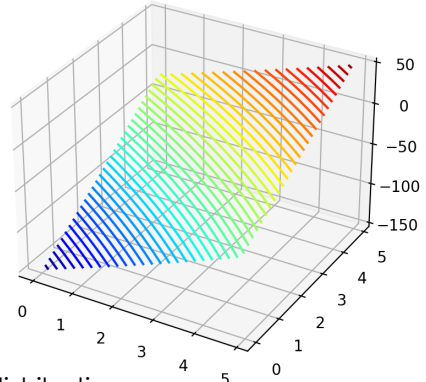
New samplers are spawned each iteration, until the test score <= 2e-7 for three consecutive iterations. (Better: stop if no new critical points found in N iterations).

Surrogate is (re)trained with a small noise injection until score <= 1e-7.

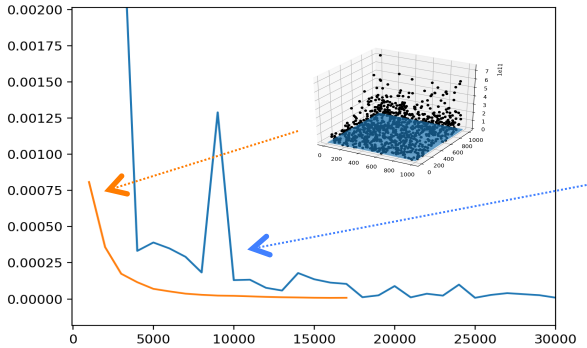




optimizer-driven sampling



sampling from a distribution



test score (on newly encountered data) converges over time

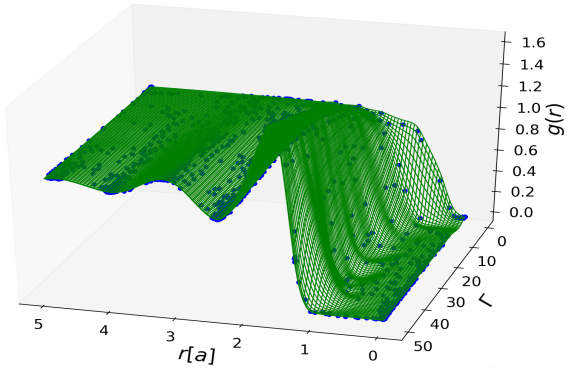
example: surrogate of one-component plasma

- We used the framework to build database and surrogate functions for OCP.
- LAMMPS is used as objective function.
- Takes 2 inputs (r, Gamma) returns g(r), self-diffusion, and viscosity

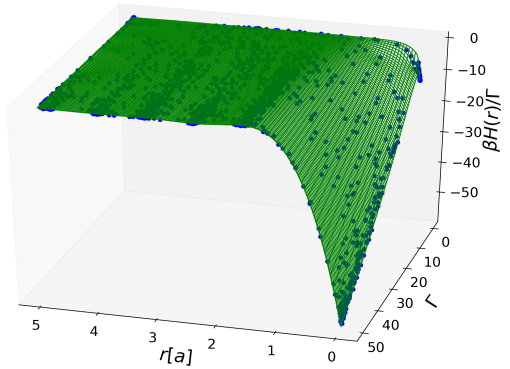
- Execution

1. Set the bounds of parameters space: r =[0, 5] and Gamma=[0,50]
2. Choose sampler → Lattice sampler
3. Choose optimizers → Nelder Mead. NCOG=10^{-4}
4. Use thin plate RBF to interpolate data
5. Tolerance for valid surrogate: max_distance=10^{-6} sum_distance=10^{-3}

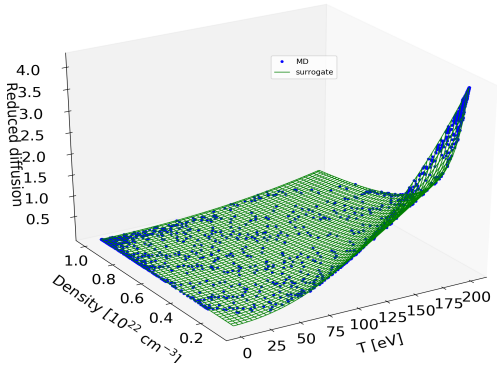
We found valid surrogates of the radial distribution function, screening function, diffusion, and viscosity after 3822 function evaluations (MD).



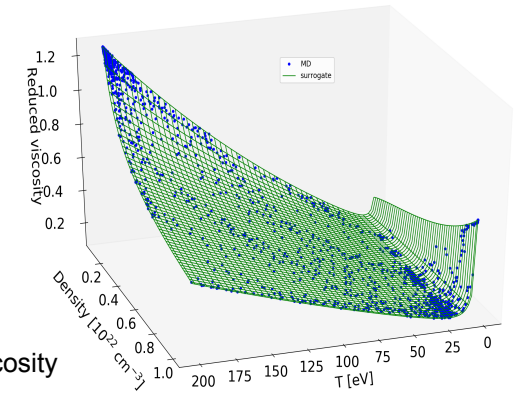
radial distribution function



screening function



diffusion



viscosity

Active Learning for Robust Particle Accelerator Beams

Objective

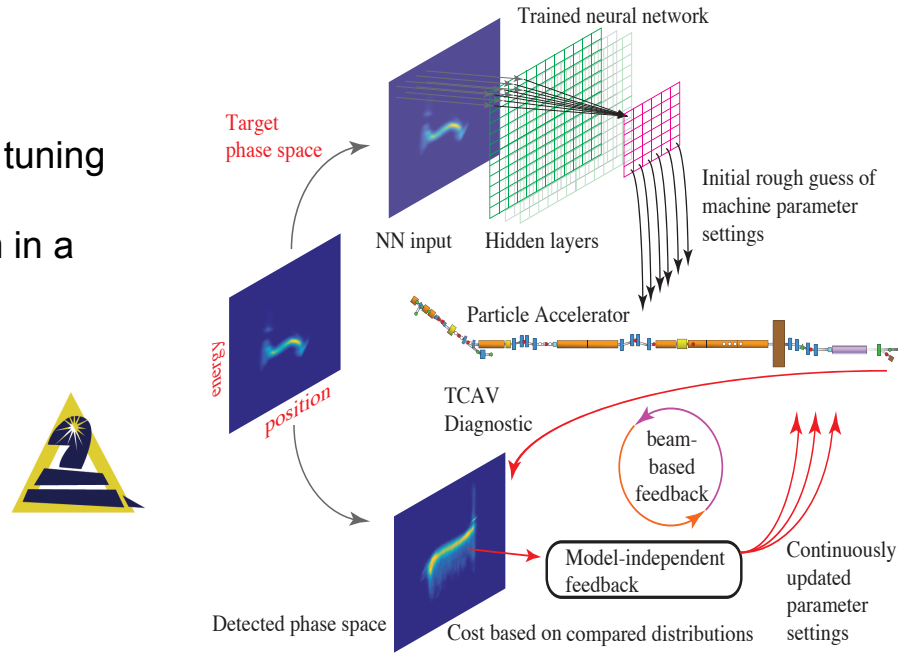
Develop adaptive feedback control tool for automated tuning of particle accelerator beams to minimize beam loss, maximize current, and provide an optimally safe beam in a dynamically changing environment[1].

Approach

Fully automate accelerator beam steering, with equipment settings as input and output indicating the quality of beam.

Train a physics-informed model that minimizes expected worst-case bounds on the quality metric between the accelerator model and data acquired from non-invasive diagnostics.

Update the learned model as newly acquired data breaks the model quality threshold.



Impact

Preliminary work[2] demonstrates longitudinal phase-space control of e^- beams at the LCLS.

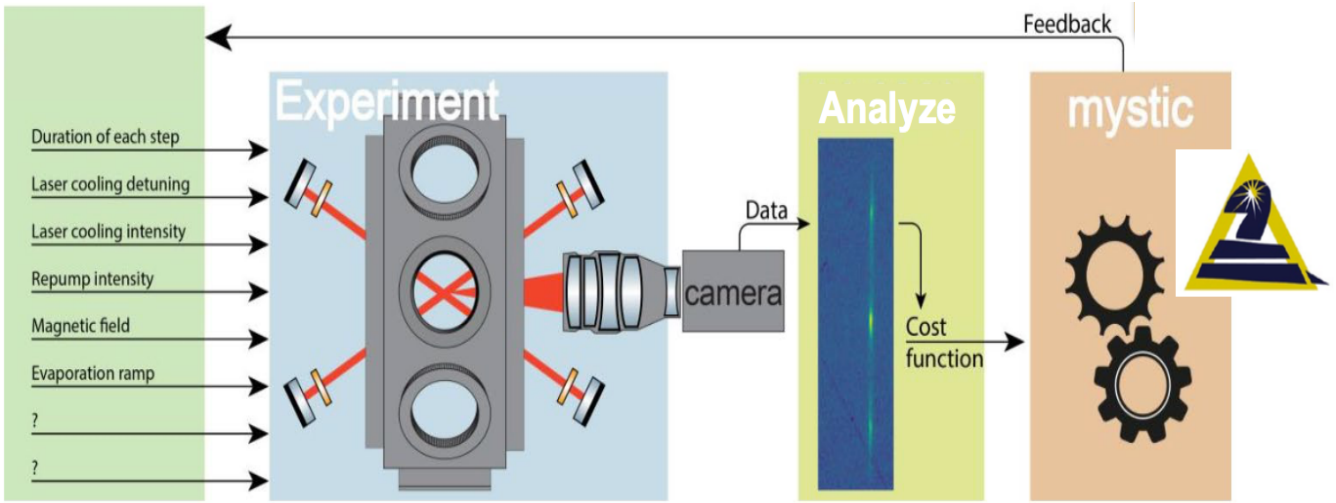
Dramatically increase rate of scientific discovery from accelerator systems, while minimizing the risk of failure and/or damage to the system.

[1] A. Scheinker 20210160DR; [2] A. Scheinker PRL, 121.4 (2018)

Active Learning for Realizing Robust Quantum Hardware

Objective

Automate learning of quantum technology with enhanced performance and robustness against environmental fluctuations[1].



Approach

Fully automate the quantum sensing experiment, with experiment settings as input and output indicating the quality of results.

Train a physics-informed model that minimizes expected worst-case bounds on the quality metric between the model and experiment.

Update the learned model as newly acquired data breaks the model quality threshold.

Impact

Preliminary work[2] indicates large performance gains for cold-atom experiments.

Expect rapid automated discovery of new capabilities to produce better manipulation of the quantum state, faster duty cycle, sensor initialization capabilities, etc.

[1] M. Martin 20210116DR; [2] A.D. Tranter Nature Com, 9 (2018)

…current work and outlook

- new optimization and learning algorithms and workflows
- new interpolation strategies and constraints/transforms
- new sampling algorithms and auto-differentiation strategies
- new auto-dimensional reductions and kernel-based sensitivities
- high-level active learning workflow improvements

- mystic is available at:
  - https://github.com/uqfoundation
- documentation and tutorials:
  - http://mystic.readthedocs.io
  - https://github.com/mmckerns/tutmom
  - https://github.com/mmckerns/tlkmys
- questions?
  - contact me at: mmckerns@uqfoundation.org

End Presentation